CS-1992-02

The Type System of a Higher-Order Logic Programming Language

Gopalan Nadathur Frank Pfenning

Department of Computer Science

Duke University

Durham, North Carolina 27708-0129

January 1992

The Type System of a Higher-Order Logic Programming Language

Gopalan Nadathur

Department of Computer Science Duke University, Durham, NC 27706

Frank Pfenning

School of Computer Science Carnegie Mellon University Pittsburgh, PA 15213

1 Introduction

The logic programming system λProlog [20] has, over the last five years, been a testbed for experimenting with several different extensions to a language such as Prolog. The initial impetus in developing this language was provided by a desire to understand the nature and role of higher-order notions within logic programming [12, 19, 21]. This goal was subsequently expanded to include devices for other forms of abstraction such as lexical scoping, modules and abstract data types [9, 10, 13]. The purpose of this paper is to discuss another important but somewhat less exposed component of the language, namely its type system.

The traditional purpose of types in programming languages has been to provide an indication, prior to execution, of errors that a program might contain. The primary form of error that typing is intended to prevent from occurring at 'runtime' is that which arises from applying a function to inappropriate arguments. An example of such an error is provided by the attempt to multiply two objects that are not numbers. The manner in which a type system functions in realizing this objective is as follows. At the very lowest level, types are utilized for partitioning the space of primitive values. These types then serve to delineate the appropriate domains for the operations of interest and also to characterize the results of such operations. Such a delineation also serves to partition the space of function values in a language that admits higher-order notions. In obtaining a practical value from such a partitioning ability, type specifications are required for the atomic expressions in the language. These are either described by the user or are obtained from the underlying programming environment. The process of type-checking then determines whether a given program satisfies the type constraints arising out of these specifications and the intended meaning of syntactic expressions. This operation is

⁰This paper is to appear in *Types in Logic Programming*, Pfenning, F. (ed), MIT Press, 1992. Comments on the paper are welcome.

performed usefully only at the time of compilation and the type errors that it unearths are in fact indications of errors that might occur at run-time if the program is left unaltered.

While the above view of the purpose of types appears to be central to most discussions of this notion, it does not constitute an exhaustive viewpoint. There has in fact been another, somewhat disparate viewpoint, that has shaped a large part of the work on the aspect of types in the context of logic programming. This viewpoint, first enunciated in [15], gives rise to what might be called the notion of descriptive types [25]. The main purpose of types under this view is as a means for characterizing particular programs that are written in a language that is, at the outset, typeless. The method used for characterizing a program within this approach has generally depended on describing the success set of each predicate that appears in it, i.e., the set of tuples of ground terms of which the predicate must be true. Such a description can be of practical utility only if it is finite and computationally 'simpler' than the program in question. This requirement has had a twofold impact. First, it has lead to an acceptance of approximations of success sets, usually in the form of their supersets. Second, it has lead to the development of languages for providing finite descriptions of sets of first-order terms. These languages are what might be called type languages: the expressions in them denote sets of terms and therefore correspond to types. The description of such languages has generally been complemented by the development of rules for inferring type annotations for the predicates that appear in programs [15, 16, 24, 25, 30].

The idea of descriptive types have several potential uses. The types that are inferred for the predicates in a program may, for instance, constitute useful information for a compiler. Inferred types may similarly be utilized as a form of documentation for a program and may also provide the programmer with an understanding of his program that is valuable in debugging. This notion of types appears, nevertheless, to be at a variance with the conventional notion of types in programming. The use of descriptive types within the framework of type-checking requires a conflation of the success set of a predicate with its domain of definition. Despite having been attempted in the past, such a conflation has the consequence of equating the concept of a program error with the notion of failure. The resulting notion of a type error is, on the one hand, far too inclusive since it negates the centrality of search within logic programming; one can well imagine the consternation of a programmer on being told that a fairly meaningful query is ill-formed because it has a negative answer¹. The notion is, on the other hand, not sufficiently inclusive since it is often desirable to consider the domain of applicability of a predicate to be distinct from its success set. A simple illustration of this fact is provided by the append predicate of Prolog. It may be intended in certain circumstances that this predicate be restricted to lists. The use of descriptive types, however, precludes

¹This situation is in contrast to a λ -calculus based language where failure, *i.e.*, inability to reduce to a normal form, may in fact be interpreted as ill-formedness.

such a restriction without the building in of a 'type-checking' mechanism into the definition of the predicate.

From the above discussion it seems clear that type-checking is useful in determining the correctness of a program only if use is made of a notion of types that is orthogonal to the idea of success sets. There is a need, in particular, to abandon the illusion of a typeless language and to adopt the so-called prescriptive notion of types. The type system of $\lambda Prolog$ is illustrative of this approach². The language of λ Prolog is in fact a strongly typed one in the sense that every well-formed expression in it is expected to have a type. The types are necessitated in the first place by the underlying logic. This logic is a higher-order logic whose consistency depends on a classification of its well-formed expressions based on types. At a pragmatic level, this requirement forces a distinction between terms denoting individuals and functions and also introduces arity requirements on function symbols. The type system has been further embellished to allow the user to declare new primitive types and to provide for a form of polymorphic typing. In the normal setting, programs and queries are presented in the context of a collection of type declarations. Since type annotations are usually omitted from programs, the process of type-checking ensures that each expression can indeed be assigned a type. Type-checking is an operation that can be performed at the time of compilation within this language and, as we shall argue, provides a valuable means for detecting program errors.

A common worry with strongly typed languages is that the need to specify typing information often intrudes on the activity of programming. The type system of the language ML indicates a means for addressing this problem. It is appropriate to think of ML as a language that is, in essence, strongly typed [17]. However, it is possible for a programmer to provide no type information with a program. There are, in general, predefined type restrictions on primitive functions and the intended semantics of the language constructs provides a means for 'reconstructing' type information for user defined functions. There are similarities in the syntax of ML and λProlog — both languages are ultimately based on a variety of typed λ -calculus — that permits this approach to be adopted in the latter language as well. The programming paradigm embodied within $\lambda Prolog$ makes the actual realization of this scheme somewhat similar in spirit to the notion type inference in the context of descriptive types. The major philosophical difference, however, is that type reconstruction in the context of λ Prolog only serves the purpose of filling in types that should have been provided in the first place; inferred types can in fact be overridden by explicit type declarations. Despite this observation, it is to be noted that type reconstruction provides a facade of typelessness in a language that is in reality typed and this turns out to be a useful feature in the process of program development.

While types are useful only in determining program correctness in other lan-

²There have also been other exemplars of this approach — see, for instance, [18] and [5].

guages, they also have a role to play in computation within λProlog . This role is twofold. First, they are useful in disambiguating overloaded operations. The mixing of polymorphism with the style of presentation of programs in λProlog in fact makes possible a form of overloading that can only reasonably be disambiguated at run-time. A consequence of this is that typing information needs to be present at run-time and in fact 'transmitted' through procedure invocations that may occur in the course of evaluating a query. Second, just as important as the type annotations of procedures is the typing information that is present within the terms of λProlog : the unification process that is used in λProlog is in reality a form of typed unification. We provide an indication in this paper of the usefulness of types in constraining unification, particularly in the higher-order context. The main point to note at this juncture is that this role of types requires them, once again, to be present at run-time. This situation is in contrast to that in most typed languages; types in these languages are required solely for the purpose of type-checking and can in fact be dispensed with during program execution.

In the remainder of this paper, we present the type system of $\lambda Prolog$ in greater detail and use this to elaborate on the above discussion. The next section describes the syntax of this language, focusing primarily on the manner in which types blend into it. The following section contains a detailed discussion of several pragmatic aspects of types in $\lambda Prolog$. In particular, we describe the notion of type-checking formally and illustrate the role that typing can play in detecting program errors. We also comment on the nature of polymorphism present in the language and explain the manner in which types interact with the notion of unification. This discussion brings out the role of types in computations and the need for them to be present at run-time, and we comment on possible situations in which they can be elided at the time of execution. The underlying assumption in Section 3 is that all the necessary type information is present in the environment, perhaps having been provided by the user. In Section 4 we elaborate on the possibility of adding type annotations when these have been omitted by the user. In particular, we describe some approaches to type reconstruction that are possible in $\lambda Prolog$ in this section.

2 The Language $\lambda Prolog$

In this section, we provide a brief overview of the logic programming language λProlog . The λProlog system has over the years been a basis for experimenting within logic programming with several structuring notions that have been found useful in other programming paradigms. The logical basis and the capabilities of this system have, therefore, been evolving. In its current and perhaps most stable form, the system is based on the logic of higher-order hereditary Harrop formulas [13]. In a more pragmatic sense, the current version of λProlog is a typed higher-order language that incorporates a form of polymorphism and includes means for

realizing notions such as abstract data types and modules.

Our focus in the discussions below is on providing an exposure to those aspects of $\lambda Prolog$ that are relevant to the rest of this paper. We begin by describing the syntax of the language and in the process reveal the fashion in which the notion of types is integrated into the overall system. We then outline the manner in which computation is performed within $\lambda Prolog$. This part of the section illustrates the similarities between Prolog and $\lambda Prolog$ and also provides the context for discussions of the role of types in computation. Finally we describe a feature of the language that in a sense lies outside its logical basis. This is the notion of type variables. Incorporating this feature into the language provides for aspects such as polymorphism and the ability to infer type information when the programmer has not provided this explicitly. We discuss these issues briefly here and postpone a fuller discussion to later sections.

2.1 The Syntax of λ Prolog Programs

The logical language that underlies λProlog is derived from Church's simple theory of types [2]. This language is typed in the sense that every well-formed expression in it has a type associated with it. The main purpose of this type is to determine the position of the expression in a functional hierarchy. At the bottommost rung of this hierarchy are what are known as atomic types. There are two sets of objects that are important in determining the atomic types: the set \mathcal{S} of sorts and the set \mathcal{C} of type constructors. Each element of \mathcal{S} constitutes an atomic type. The type constructors are each specified with a unique arity and may be used in obtaining other atomic types as follows: if $c \in \mathcal{C}$ is of arity n and c_1, \ldots, c_n are types, then c_1, \ldots, c_n is an atomic type. The other types in the hierarchy are the c_1, \ldots, c_n is a type, corresponding to the set of functions whose domain and range are given by c_1 and c_2 respectively.

This rather abstract description of types is specialized to λProlog in the following way. The set \mathcal{C} is initially empty and the set of sorts contains only int, the type corresponding to the integers, and o, the type of propositions. The user of the system is permitted to add new type constructors by using a declaration of the form

kind
$$c$$
 type $\rightarrow \dots \rightarrow$ type.

The arity of the constructor c that is thus declared is one less than the number of occurrences of type in the declaration. Noting that a sort may be viewed as a nullary constructor, a declaration of the above kind may also be used for adding new sorts. As a specific example, the declarations

kind i type.
kind list type
$$\rightarrow$$
 type.

add i to the set of sorts and define list as a unary constructor. Using list, several new atomic types such as $(list\ i)$, $(list\ int)$ and $(list\ (i \rightarrow i))$ might now be obtained. There are, of course, an infinite collection of function types that may be constructed using these atomic types. Examples of such types are

```
\begin{array}{ll} \operatorname{int} \to \operatorname{int} & \operatorname{corresponding} \ \operatorname{to} \ \operatorname{functions} \ \operatorname{on} \ \operatorname{integers}, \\ \operatorname{int} \to \operatorname{o} & \operatorname{corresponding} \ \operatorname{to} \ \operatorname{predicates} \ \operatorname{on} \ \operatorname{integers}, \ \operatorname{and} \\ \operatorname{int} \to (\operatorname{int} \to \operatorname{int}) & \operatorname{corresponding} \ \operatorname{to} \ \operatorname{functions} \ \operatorname{from} \ \operatorname{integers} \\ & \operatorname{to} \ \operatorname{functions} \ \operatorname{on} \ \operatorname{integers}. \end{array}
```

Lest the readings accorded to the various types turn out to be misleading, it should be pointed out that no type constructor has an *a priori* meaning in λ Prolog and the only atomic type that has an initial interpretation is o.

The use of parentheses can be reduced in $\lambda Prolog$ by adopting the convention that \to is right associative. For example, the last of the function types displayed above can be written as $int \to int$. Using this convention, every function type can in fact be depicted by an expression of the form $\tau_1 \to \ldots \to \tau_n \to \tau$ where τ is an atomic type. The argument types of a function type correspond to the τ_i 's and the target type corresponds to τ when it is depicted in this form. By an abuse of notation, we shall use the above representation even in a situation where n is 0, thereby permitting the terminology to be extended to all types.

The *terms* of the language are constructed from given sets of constant and variable symbols, each of which is assumed to be specified with a type. The constants are categorized as the *logical* and the *nonlogical* ones. In the context of $\lambda Prolog$, the logical constants consist of the following:

```
true denoting the true proposition, , of type o \to o \to o, representing conjunction, ; of type o \to o \to o, representing disjunction, \leftarrow of type o \to o \to o, sigma of type (\tau \to o) \to o for each type \tau, pi of type (\tau \to o) \to o for each type \tau.
```

The symbols sigma and pi in reality stand for a family of constants, each parameterized by the choice of τ . For the moment we shall assume that the particular constant intended can be determined from the context. The notion of type variables introduced later in this section will permit these constants to be interpreted as polymorphic ones, in a manner consistent with this assumption.

The tokens representing the other constants and variables in λ Prolog are identified in much the same way as in Prolog, *i.e.*, they correspond to sequences of alphanumeric characters or sequences of 'sign' characters. In addition, those tokens that begin with uppercase letters are distinguished as variables. Each of these

symbols in λ Prolog has a type. For those symbols that consist solely of numeric characters, the type is assumed to be int. The type of other constants is given by declarations of the form

```
type constant type-expression.
```

The import of such a declaration is to identify the type of *constant* with the corresponding type expression. As an example, the declarations

```
type nil list int.

type :: int \rightarrow (list int) \rightarrow (list int).
```

define the constants nil and :: that function as constructors of lists of objects of type int. Types of constants and variables may also be indicated by writing them in juxtaposition and separated by a colon. Thus the notation X:int corresponds to a variable X of type int. For constants, typing information that is provided in this manner has a different effect from that of the type declaration just discussed. The precise nature of this difference will become clear in Section 3.

The terms of λ Prolog are obtained from the constant and variable symbols by using the mechanisms of function abstraction and application. In particular

- 1. each constant and variable of type τ is a term of type τ ,
- 2. if X is a variable and T is a term of type τ' , then $(X : \tau \setminus T)$ is a term of type $\tau \to \tau'$, and
- 3. if T_1 is a term of type $(\tau_2 \to \tau_1)$ and T_2 is a term of type τ_2 , then $(T_1 \ T_2)$ is a term of type τ_1 .

The term obtained by virtue of (2) is referred to as an abstraction whose bound variable is X and whose scope is T. Similarly the term obtained by (3) is called the application of T_1 to T_2 . Using the constant :: defined above, the following expression constitutes an example of a term:

```
((X : int) \setminus ((:: X) (Y : list int))).
```

It is in fact a term of type $int \rightarrow (list \ int)$.

There are several devices provided in $\lambda Prolog$ for simplifying the presentation of expressions. For example, parentheses may be omitted by using the convention that application is left associative. Similarly, constants may often be written as operators. Thus, the constants \leftarrow , \Rightarrow , ;, and , are assumed by convention to be infix operators. In addition, the user may specify constants that need to be viewed as operators. For instance, the following declaration

infix
$$225$$
 xfy :: .

functions in the same way as does the declaration op(225, xfy, ::) in Prolog; it defines :: to be a right associative infix operator of precedence 225. Finally, when the types of constants or variables in an expression can be uniquely determined by assuming the expression is well-formed, these can be omitted. Using the various conventions outlined and the infix declaration above, the term displayed earlier can be abbreviated by

$$Y \setminus (X \setminus (X :: Y)).$$

Given the type of ::, the types of the variables X and Y in this expression can be determined to be int and $(list\ int)$.

We now identify expressions in the language that are relevant to writing programs in $\lambda Prolog$. To begin with, we define the *atomic formulas* to be the terms of type o that have the structure $(P \ T_1 \ \dots \ T_n)$ where P is either a nonlogical constant or a variable. The atomic formula is referred to as a rigid atom in the case when P is a nonlogical constant, and as a flexible atom otherwise. Using the symbol A to denote an arbitrary atom and A_r to denote a rigid atom, the classes of G- and D-formulas may now be defined by mutual recursion as follows:

$$G ::= true \mid A \mid (G_1, G_2) \mid (G_1; G_2) \mid sigma (X : \tau \setminus G) \mid pi (X : \tau \setminus G) \mid (D \Rightarrow G)$$

$$D ::= A_r \mid A_r \leftarrow G \mid pi (X : \alpha \setminus D) \mid (D_1, D_2)$$

As will become clear when the notion of computation is discussed, the symbols \Rightarrow and \leftarrow are intended to correspond to implication with it being written 'backwards' in the latter case. The symbols pi and sigma are similarly meant to represent universal and existential quantification respectively. The quantifiers that are used in conventional presentations of logic play a dual role: in the expression $\forall x \ P(x)$, the quantifier has the function of binding the variable x over the expression P(x) in addition to that of making a predication of the result. In our system, these roles are separated between the abstraction operation and appropriately chosen constants. Thus the expression $\forall x \ P(x)$ is in fact represented by one of the form $(pi\ (X\setminus (P\ X)))$. The constant sigma plays a similar role with regard to existential quantification.

The G- and D-formulas determine the programs and queries of $\lambda Prolog$ as follows. A program consists of a set of (implicitly universally closed) D-formulas each element of which is referred to as a program clause, and a query is a (implicitly existentially closed) G-formula. We shall depict programs by writing a sequence of program clauses, each clause being terminated by a period. Similarly, a G-formula followed by a question mark will symbolize a query. As a specific example of a program, one might consider the following, assuming member has been declared to be a constant of type $int \rightarrow (list\ int) \rightarrow o$:

member
$$X (X :: L)$$
.

```
member X (Y :: L) \leftarrow \text{member } X L.
```

A query in the context of this program may then be

```
member 1 (3 :: 2 :: 1 :: nil)?
```

A point to note from these examples is the similarity of the λ Prolog syntax for programs and queries to that of Prolog. The one major difference to be observed is the use of a curried notation.

The use of implicit quantification in program clauses and queries raises the possibility of a problem with regard to the declaration of types for variables. This problem may be overcome by using the notation $X:\tau$ at one of the occurrences of the variable to indicate its type. Thus an alternative presentation of the first clause in the program above might have been

```
member (X : int) (X :: (L : (list int))).
```

In this particular case the indication of types for X and L are really unnecessary since their values are determined uniquely by the context, but the notation may be useful in other situations. The extent to which the type annotation may be omitted altogether is a matter that will concern us in Section 4.

It has been found useful to organize declarations into modules and this has therefore been introduced as a structuring concept within $\lambda Prolog$. Modules are, in this context, named environments that have various kind, operator and type declarations as well as program clauses associated with them. As an example of this structure, the following collection associates the name lists with the declarations presented at various places in this section:

```
module
               lists.
kind
               list
                              type \rightarrow type.
infix
               225
                              xfy
               nil
                              (list int).
type
type
                              int \rightarrow (list int) \rightarrow (list int).
               ::
type
               member
                              int \rightarrow (list int) \rightarrow o.
member X(X :: L).
member X(Y::L) \leftarrow \text{member } XL.
```

A module may enlargen the set of declarations available within it by *importing* other modules. The effect of this operation it to make available within the module being defined the definitions contained in the module being imported. A logical characterization of the effect of this operation insofar as program clauses are concerned may be provided via the notion of implication (see, for instance, [10]), but this aspect will not concern us here. An aspect that is of interest is that the module boundary provides a notion of scope with regard to declarations. This notion is

useful, for instance, in the process of type reconstruction, an issue that is considered in Section 4.

2.2 The Nature of Computation

A computation in λ Prolog involves constructing a derivation for a given query from a collection of program clauses. Prior to describing this process, the notion of an instance of a program clause is required. This may be outlined as follows: (i) if A'_r and G' are obtained from A_r and G by replacing the free variables by new ones, then $(A'_r \leftarrow G')$ and A'_r are, respectively, instances of $(A_r \leftarrow G)$ and A_r , (ii) the instances of D_1 and D_2 are instances of (D_1, D_2) , and (iii) the instances of D are instances of (P_1, P_2) , and (iii) the instances of P_1 are instances of P_2 are instances of a computation may be explicated by describing the possible next steps at any intermediate stage in the computation. Assuming that the stage in question corresponds to constructing a derivation for the query P_1 from the program P_2 , the computation may proceed to one of the following steps depending on the structure the query:

- 1. to constructing a derivation for $\theta(G')$, if G is a rigid atom and $A \leftarrow G'$ is an instance of a program clause in \mathcal{P} such that G can be 'unified' with A by the substitution θ ,
- 2. to constructing a derivation for either G_1 or G_2 , if G is $(G_1; G_2)$,
- 3. to constructing a derivation for both G_1 and G_2 , if G is (G_1, G_2) (observing that the instantiations determined for G_1 and G_2 in these derivations must be consistent),
- 4. to constructing a derivation for G'_1 if G is $(sigma\ (X \setminus G_1))$, where G'_1 is obtained from G_1 by replacing X by a new variable,
- 5. to constructing a derivation for G'_1 if G is $(pi\ (X \setminus G_1))$, where G'_1 obtained from G_1 by replacing X by a new constant symbol, and
- 6. to constructing a derivation for G' with the program clauses augmented with D if G is $(D \Rightarrow G')$.

A derivation can be terminated successfully if the query in question is either true or a flexible atom of the form $(P \ T_1 \ ... \ T_n)$ or a rigid atom that is 'unified' by the substitution θ with an instance of a program clause in \mathcal{P} ; in the latter two cases, the termination also determines the substitutions $\{\langle P, (X_1 \ ... \ X_n \ true) \rangle\}$ and θ for the variables in the query. When a derivation terminates successfully, the result of the computation can, as usual, be provided by a substitution that is determined by the derivation.

There are certain points to be observed with regard to the above, somewhat informal, description of a λ Prolog derivation. First, the process of constructing a

derivation is actually a non-deterministic one. In particular, there are choices to be exercised with respect to program clause instances, unifying substitutions and disjunct to be derived in a disjunctive goal. Second, the notion of unification is a great deal more involved than the one of relevance in the context of Prolog. To begin with, the requirement here is to unify typed λ -terms in a situation where we consider two terms to be equal if they can be made identical by the rules of λ -conversion [6]. A unification procedure of this sort is described in [7]. In addition, there are restrictions on the kinds of terms that might be substituted for variables arising from the fact that queries may contain universal quantifiers. As a particular example, the unification procedure used in conjunction with the above notion of a derivation must preclude a successful derivation from being found for $sigma~(X \setminus (pi~(Y \setminus (p~X~Y))))$ from the program $\{(p~X~X)\}$. Although an appropriate unification procedure can be defined formally [11, 22], we do not discuss it further here. For our purposes, only an informal understanding of the nature of computation is required.

From the informal description, it is clear that there is much in common between computations in λ Prolog and in Prolog. Consider, for instance, a derivation for the query

```
member 1 (3 :: 2 :: 1 :: nil)?
```

given the program defining member that was presented earlier. This derivation would require constructing subderivations for the formulas

```
member 1 (2 :: 1 :: nil) and member 1 (1 :: nil)
```

in turn. The last query obviously succeeds, thereby leading to a successful derivation overall. This is, of course, similar to the manner in which a derivation might have been constructed in Prolog. The only, somewhat unnoticeable, difference is that the program under consideration in λ Prolog is typed.

Some of the differences with Prolog might be brought out by considering the following declarations in the context of the type declarations in the module *lists*:

```
type mapfun (int \rightarrow int) \rightarrow (list int) \rightarrow (list int) \rightarrow o. mapfun F nil nil. mapfun F (X :: L1) ((F X) :: L2) \leftarrow mapfun F L1 L2.
```

There is a function variable, F, that appears in the program clauses above. Such variables are, of course, not permitted in Prolog. One difference that such variables make is that they allow computations to be performed via the notion of reduction. As a particular example, consider the query

```
mapfun (X \setminus (g \mid 1 \mid X)) \mid (1 :: 2 :: nil) \mid L?
```

in a context where the type of g is declared to be $(int \to int \to int)$. Evaluating this query results in the value $((g\ 1\ 1)\ ::\ (g\ 1\ 2)\ ::\ nil)$ being computed for L. In obtaining this answer, two reductions need to be done: $((X\setminus (g\ 1\ X))\ 1)$ and $((X\setminus (g\ 1\ X))\ 2)$ must be reduced to $(g\ 1\ 1)$ and $(g\ 1\ 2)$ respectively.

In the above example, the function variable was effectively instantiated by a term in the query. However, it is not necessary that such variables be always instantiated in this fashion: values can in fact be computed for them through unification. Consider for instance the evaluation of the query

```
mapfun F (1 :: 2 :: nil) ((g 1 1) :: (g 1 2) :: nil)?
```

This query would require the value $(X \setminus (g \ 1 \ X))$ to be found for F. Tracing through the computation involved, it can be observed that a unifier for $(F \ 1)$ and $(g \ 1 \ 1)$ would first need to be computed. In a situation where equality of terms incorporates the notion of λ -conversion, there are four incomparable 'most general' unifiers for these terms. These are given by the substitutions $(X \setminus (g \ 1 \ 1)), (X \setminus (g \ 1 \ X)), (X \setminus (g \ X \ 1))$ and $(X \setminus (g \ X \ X))$ for F. At a subsequent stage in the computation, a unifier for the terms $(F \ 2)$ and $(g \ 1 \ 2)$ will have to be found, and only one of these substitutions is satisfactory for this purpose as well.

As a final aspect, we observe that the λ Prolog syntax permits predicates to be variables as well. An illustration of a program using this feature is the counterpart to mapfun that is defined below:

```
type mappred (int \rightarrow int \rightarrow o) \rightarrow (list int)\rightarrow o. mappred P nil nil. mappred P (X :: L1) (Y :: L2) \leftarrow P X Y, mappred P L1 L2.
```

As in the case of mapfun, this 'procedure' can be invoked by instantiating the predicate argument. Doing so could result in a computation that is more involved than the operation of β -reduction in the simply typed λ -calculus. We note in particular the presence of the goal $(P \ X \ Y)$ in the body of the clause. Given that the predicate variable P can be instantiated by a term that contains logical constants in it, we see that a complex goal may actually need to be evaluated. It follows from this that more meaningful computations can be performed by running mappred in, so to speak, a forward fashion. However, this feature also ensures the difficulty of finding appropriate substitutions for predicate variables when these appear as the heads of goals, *i.e.*, in what might be called an extensional position. It turns out that a rather simple substitution always suffices and it is such a substitution that is returned in $\lambda Prolog$. For instance, a query such as

```
mappred P (1 :: 2 :: nil) (3 :: 4 :: nil)?
```

would produce the value $(X \setminus Y \setminus true)$ for P even though several more imaginative results could be conceived of. It should be noted that this aspect does not trivialize

the use of predicate variables. To begin with, several capabilities of higher-order programming in functional programming contexts can be realized merely by the ability to instantiate predicate variables. Further, there is the possibility of utilizing intensional occurrences of predicate variables in finding meaningful substitutions for them that then give rise to complex computations by virtue of extensional occurrences of the same predicate variables. A more detailed discussion of such uses of predicate variables may be found in [19].

2.3 Type Variables

A closer look at the declarations encountered so far reveals that the use of 'simple' types makes the definitions of procedures more specific than they need to be. Thus, the predicates member, mapfun and mappred as defined above pertain only to lists of integers even though the structure of the defining clauses would remain identical for lists of other kinds of objects. As a particular example, if we identify the constants bob, sue, etc. as being of type person, we would, under the present circumstances, need another definition of the member predicate for determining membership of these objects in lists of persons.

This situation is clearly unsatisfactory from a programming point of view since it leads to a proliferation of declarations, and to a complete separation between notions that at some level have a common flavor. An immediate solution to the above problem is to collapse the type distinctions between terms that have an atomic type. Thus we may agree by convention that the only type other than o is i, and then deem that both bob and 3 are of type i; in a sense this is the course adopted in Prolog. This is not a very happy solution to the given problem because there are pragmatic reasons for maintaining distinctions between terms of different atomic types as we shall argue in the next section. Even if we were to accept the suggested solution to the above problem, we notice that it is certainly not a solution if what we desire is one definition of the procedures in question that will be satisfactory for lists of terms of each function type as well. The distinction between the types $i \rightarrow i$ and i, for instance, is intrinsic to the logical language on which λProlog is based.

The solution that we have adopted to the problem is to allow variables in types. This solution constitutes a reflection of a syntactic device that is often employed in discussions about the logical system into the language itself. An illustration of this approach is provided by the revised version of the module *lists* that appears below; we note that the tokens that begin with capital letters in the types in the type declarations stand for variables.

```
type member A \rightarrow (list \ A) \rightarrow o.
member X \ (X :: L).
member X \ (Y :: L) \leftarrow member \ X \ (X :: L).
```

A type declaration in which variables occur in the type is to be understood in the following fashion: It represents an infinite number of declarations each of which is obtained by substituting, in a uniform manner, closed types for the variables that occur in the type. For instance, the type declaration

type ::
$$A \rightarrow (list A) \rightarrow (list A)$$
.

represents, amongst others, the type declarations

type ::
$$\operatorname{int} \to (\operatorname{list\ int}) \to (\operatorname{list\ int}).$$
type :: $(\operatorname{int} \to \operatorname{int}) \to (\operatorname{list\ (int} \to \operatorname{int})) \to (\operatorname{list\ (int} \to \operatorname{int})).$

As another instance, observe that the family of logical constants sigma and pi can be thought of as being represented by the declarations

$$\begin{array}{lll} \text{type} & \text{sigma} & (A \rightarrow o) \rightarrow o. \\ \text{type} & \text{pi} & (A \rightarrow o) \rightarrow o. \end{array}$$

Such a treatment is, in fact, the one accorded to them in $\lambda Prolog$.

By virtue of the above provision, we see that variables may occur in the types corresponding to the constants and variables that appear in a program clause. Such a clause is, again, to be thought of as a schema that represents an infinite set of procedure declarations in which no type variables appear. Each member of this set is obtained by substituting closed types for the type variables that occur in the schema. Such a substitution is, of course, constrained by the fact that the resulting instance must be a well-formed program clause. Thus, consider the program clause 'schema'

member
$$(X : A1) (X :: (L : (list A2)).$$

Writing the types of member and :: as $A3 \rightarrow (list\ A3) \rightarrow o$ and $A4 \rightarrow (list\ A4) \rightarrow (list\ A4)$ respectively, the same type must replace the variables $A1,\ A2,\ A3$ and A4 in all the permissible instances of this schema.

It is tempting to conclude that the use of type variables in λProlog actually provides for an enriched class of program clauses in which an (implicit) universal quantification is provided over type variables that occur in each clause. While this view is correct at an intuitive level, it is not mirrored precisely within Church's simple theory of types, the logic underlying λProlog : this theory has no provision for explicit quantification over type variables. However, one can make sense of clauses in which these variables occur within this logic by interpreting them as schemas standing for infinite sets of clauses. One implication of this is that, in order to

be logically correct, the invocation of a program clause must also be accompanied by a determination of the type instance to be used. In practice this choice is delayed through unification at the level of types. We shall comment further on this possibility in Section 3.

The introduction of type variables permits much of the burden of providing type declarations to be removed from the user. An adequate set of type declarations can be inferred by using the methods described by [14] in the context of ML [4]. As an illustration, consider the following program clause:

```
member X (Y :: L) \leftarrow \text{member } X L.
```

Given the type of ::, the type $A \to (list\ B) \to o$ can be inferred for member. The idea is that the requirement that type instances of a program clause be well-formed places constraints on the types that may be associated with the constants and variables that appear in it. These constraints may be formulated as a system of equations involving first-order terms that may then be solved by using the notion of (first-order) unification [27] to obtain a set of implicit type declarations. Unfortunately these constraints are not sufficient for determining types uniquely. For example, given the clauses

```
member X (X :: L).
member X (Y :: L) \leftarrow member X L.
```

member might be given types $A \to (list \ A) \to o$ or $A \to B \to o$ or $int \to (list \ int) \to o$, etc. This issue will be discussed further in Section 4; for the moment we observe that the theory dictates only that member have a type and the choice between different possibilities must be based on pragmatic and philosophical considerations. Further, the user may override such a choice by providing an explicit type declaration.

While the syntax of types and the methods of inferring the types associated with terms in λProlog bears a resemblance to that used in the context of functional programming languages (specifically ML), it is important to point out that there is a difference with regard to the role that they play in the overall system. The primary purpose of typing in ML is to provide a partial assurance of the correctness of a program and is in a certain sense orthogonal to the evaluation mechanism: As [14] argues, a program that can be well-typed is semantically sound. Looked at operationally this means that, for example, "an integer is never added to a truth value or applied to an argument, and consequently need not carry its type around for run-time checking." Types make a similar contribution to program checking in λProlog and the task of type-checking can for the most part be dispensed with at 'compile-time.' However, in contrast to the situation in ML, types also play a significant role in the evaluation mechanism underlying λProlog and consequently do not constitute a component that is dispensable at run-time. Instantiations determined

for type variables must in fact be transmitted during execution. These aspects and others are discussed in greater detail in the next section.

3 The Role of Types in the System

The discussions of the previous section make clear the centrality of the notion of types in the syntax of λ Prolog. However, a matter that bears further elucidation is the reason for including types in the language as also the role of types within the overall system. At the very outset, a simple justification can be provided for the presence of types: the consistency of the logical language that underlies $\lambda Prolog$ depends on a classification of its well-formed expressions. It should be readily apparent that the logical language that is of interest is one that is obtained by superimposing a logic of connectives and quantifiers over the understanding of functions provided by the λ -calculus. The first attempt at describing such a logic utilized the untyped λ -calculus as a starting point. Unfortunately this attempt did not come to fruition: assuming even the most rudimentary properties of the logical connectives permitted the well-known set theoretic paradoxes to be recreated in this context. For example, it was shown by Curry that Russell's paradox could be mirrored in a situation where only apparently essential properties were assumed for the implication symbol (see, for instance, [6]). One approach to overcoming this difficulty turns out to be that of introducing a hierarchical organization over the expressions of the language in the spirit of Russell. This observation has, in fact, been the reason for developing logics based on typed λ -calculi.

This simple argument for the presence of types is, however, not quite satisfactory given that our primary concern is that of a programming language. We attempt to provide an alternative justification by describing the manner in which types interact with the task of writing programs in λ Prolog. Our first concern in this direction is that of elaborating on the notion of type-checking in this language. We then outline the function of types in making distinctions between objects and illustrate some of the consequences of such distinctions. In particular, we argue that a properly chosen set of types can aid both in the clarity and correctness of programs; towards demonstrating the latter we illustrate the use of type-checking in discovering program errors. We then discuss the nature of the polymorphism present in the language. We show here that there is in fact a curious mix of ad hoc and parametric polymorphism in λ Prolog. Finally we discuss the role of types in unification. From this discussion it becomes clear that types are needed in determining computations and are therefore a necessary component of the run-time environment in $\lambda Prolog$. There is, nevertheless, a possibility of eliding them at run-time and we touch on this aspect briefly.

3.1 The Notion of Type-Checking

The logical language that underlies λProlog requires certain type constraints to be satisfied by expressions that are well-formed. Thus, an expression of the form (T_1, T_2) is well-formed only if T_1 and T_2 have types of the form $\tau_2 \to \tau_1$ and τ_2 , respectively. The issue of type-checking amounts to determining whether such constraints are satisfied by any given expression. In the typical scenario, types are only specified with the atomic parts of an expression. The process of type-checking therefore also involves the assignment of types to well-formed subparts of the expression. We describe an inference system below that enables these issues to be addressed simultaneously. In particular, the inference system will serve to determine if an expression satisfies typing constraints given the types of the atomic symbols appearing in it as well as to find the type to be assigned to the expression if this is the case.

In order to describe the desired inference system, it is necessary to make the informal understanding of type variables in λ Prolog somewhat more formal. For this purpose, we distinguish between two categories of expressions, the *simple types* and the *type schemas*. The simple types correspond to the type expressions that are legal within λ Prolog. In particular, their syntax is given by the rule

$$\tau ::= \alpha \mid s \mid (c^{(k)} \tau_1 \ldots \tau_k) \mid (\tau_1 \to \tau_2)$$

where α ranges over type variables, s ranges over sorts and $c^{(k)}$ ranges over k-ary type constructors. The type schemas are obtained by permitting an explicit quantification over the type variables in a simple type as follows:

$$\sigma ::= \tau \mid \forall \alpha. \ \sigma.$$

We use τ for simple types and σ for type schemas in the discussions below. The quantification present in type schemas is not available within the syntax of $\lambda Prolog$. It is, nevertheless, needed in order to bestow the right interpretation on constants whose type declarations contain type variables. For instance, the declaration

type member
$$A \to (\text{list } A) \to o$$
.

is to be construed as associating the type schema $\forall A. (A \rightarrow (list \ A) \rightarrow o)$ with member. In accordance with the previous discussion of type variables, member, by virtue of this declaration, is to be interpreted as a family of constants indexed by the instantiation for A.

As mentioned already, our type-checking calculus will determine if a given expression is well-typed under assumed typings for the atomic symbols appearing in it. Different treatments will be accorded to variables and constants within this calculus, type schemas being permitted only with the latter category of symbols. This distinction is manifest in the separation of the assumed typings into *contexts*? of

the form $X_1:\tau_1,\ldots,X_n:\tau_n$, assigning simple types to variables, and signatures Σ of the form $c_1:\sigma_1,\ldots,c_n:\sigma_1$ assigning type schemas to constants. We assume that each variable and constant appears at most once in a context and signature, respectively. Furthermore, we assume that the types appearing in contexts and signatures are always well-formed, which means that all appearing type constructors are used with the correct number of arguments.

We wish to associate, via the process of type-checking, a unique type with every expression that is well-formed. At the very outset, this requires a syntax that ensures unique types are associated with each atomic symbol. Unfortunately the syntax of λ Prolog as discussed to this point does not satisfy this requirement. The particular problem is with constants — the association of type schemas with these symbols leaves their type ambiguous. Our solution to this problem is to adorn constants with a list of indices that determine the type instantiation desired. Thus, we may use $member^{int}$ to represent that element of the family corresponding to member whose type is $int \rightarrow (list\ int) \rightarrow o$. Notice that this device is not directly available in λ Prolog, but its effect may be obtained through explicit type annotations of the form $T:\tau$.

We now describe precisely the syntax of expressions whose type our calculus will check. We call these expressions pre-terms to highlight the fact that they may not actually be terms because typing requirements may be violated. Our desire to focus on the issue of typing prompts a somewhat simplistic syntax for pre-terms, but one that still reflects the commitment to an unambiguous language that is discussed above. In particular, we shall assume that pre-terms are given by the rule

$$T \quad ::= \quad X \mid c^{\tau_1, \dots, \tau_n} \mid (T_1 \mid T_2) \mid ((X : \tau) \setminus T) \mid T : \tau$$

in which X ranges over variables and c ranges over constants. This syntax is simplistic in that it makes no distinction between terms, queries and program clauses in $\lambda Prolog$. However, the simplification is of an inessential nature: the distinctions between the three categories of expressions can be obtained by placing appropriate restrictions on the appearance of logical constants. In the interest of uniformity of presentation, we obscure these distinctions for the moment.

We provide in Figure 1 a set of inference rules that can be used to check the type of a pre-term. The judgements that can be made using this calculus take the form of assertions written as ? $\vdash_{\Sigma} T : \tau$, where T is a pre-term and τ is a simple type. We use the notation $[\tau'/\alpha]\sigma$ here to denote the type schema that results from the substitution of τ' for α in σ . Observe that these rules incorporate a polymorphic view of constants by permitting their type schemas to be instantiated at each occurrence where we require a simple type for them.

The assumption of unique types for atomic symbols that was made above was necessary to ensure an unique type for each well-formed expression and, ultimately, an unambiguous semantics for clauses and queries. In practice, $\lambda Prolog$ syntax

$$\frac{X : \tau \quad \text{in} \quad ?}{? \vdash_{\Sigma} X : \tau} \qquad \frac{c : \forall \alpha_{1} \dots \forall \alpha_{n}, \tau \quad \text{in} \quad \Sigma}{? \vdash_{\Sigma} c^{\tau_{1}, \dots, \tau_{n}} : [\tau_{n}/\alpha_{n}] \dots [\tau_{1}/\alpha_{1}]\tau}$$

$$\frac{? \vdash_{\Sigma} T_{1} : \tau_{2} \to \tau_{1} \qquad ? \vdash_{\Sigma} T_{2} : \tau_{2}}{? \vdash_{\Sigma} (T_{1} T_{2}) : \tau_{1}} \qquad \frac{? , X : \tau' \vdash_{\Sigma} T : \tau}{? \vdash_{\Sigma} (X : \tau' \setminus T) : \tau' \to \tau}$$

$$\frac{? \vdash_{\Sigma} T : \tau}{? \vdash_{\Sigma} (T : \tau) : \tau}$$

Figure 1: The Typing Rules

permits the writing of partially typed terms. In analogy to the pre-terms, the syntax of these terms may be given by the rule

$$T \quad ::= \quad X \mid c \mid (T_1 \mid T_2) \mid ((X : \tau) \setminus T) \mid (X \setminus T) \mid T : \tau$$

The major difference, of course, is that explicit types at occurrences of constants and variable declarations may be omitted. The issue of type-checking as such cannot be meaningfully raised for these expressions. However, one can ask a related question, i.e. that of typeability: given an expression T in this syntax, does there exist a pre-term T' that can be obtained by adding types at constant occurrences and variable declarations and that is well-typed according to the typing rules in Figure 1? Besides the question of typeability, there is another question that is of interest, namely that of type reconstruction: if there are multiple ways to restore types, is there a particular one that the compiler should choose? Since types affect the meanings of programs, this question has a direct bearing on the issue of what the desired semantics for a partially typed program is. Given that λP rolog programs are generally only partially typed, the question of type reconstruction is an important one, and we discuss it at length in Section 4.

3.2 Types as a Means for Detecting Program Errors

From a theoretical perspective, the primary purpose of types is to make distinctions between terms based on a functional hierarchy. Thus, there is a difference between terms denoting individuals and those denoting functions, and typing is intended to bring this out. The need for such a distinction can be justified from a pragmatic perspective as well. Consider, for instance, the following clause

mapfun F (X :: L1) ((F X) :: L2)
$$\leftarrow$$
 mapfun F L1 L2.

It is useful to think of the variable F as a function variable, since it must be possible to meaningfully 'apply' it to objects. On this premise, the following query

```
mapfun 2 (1 :: nil) L?
```

may be ruled as ill-formed: it is meaningless to apply the integer 2 to the integer 1 under the usual understanding of integers. A typing scheme that distinguishes between functions on integers and integers permits such an 'error' to be detected the moment the query is written down, *i.e.*, at compile-time. An additional benefit of typing is that it provides useful documentation of a program. Thus, the fact that the first argument of mapfun must be a function conveys additional information about the predicate.

It is relevant to note that the distinction discussed above is meaningful only in the context of a higher-order language: it is only in this situation that there is a need to distinguish between terms denoting functions and terms denoting individuals. Now in principle it is possible to collapse all other distinctions within the type system of λ Prolog. Thus, we may deem that there is only one sort i (or int, since this is a built-in sort), and that every term denoting an individual object is of this type. Thus, the types of some familiar constants may be identified as follows:

```
2: i "abc": i nil: i :: i \rightarrow i \rightarrow i append: i \rightarrow i \rightarrow i \rightarrow o
```

The types of some of these constants might be contrasted with those provided in the previous section. We observe that if higher-order terms are excluded, the typing indicated here is close to being redundant and hence similar to the one in 'effect' in Prolog. The main difference is Prolog does not even enforce arity requirements on function and predicate symbols.

Although type distinctions between first-order terms can be collapsed, the type system of λProlog facilitates such distinctions. The provision of such a feature is justified only if there is some programming use for it. It turns out that the ability to distinguish between individual terms is quite useful from the perspective of ensuring the clarity and correctness of programs. For example, consider the expression (2:: "abc") in the context of the above typing. This expression turns out to be well-typed and is therefore an acceptable term. This situation is somewhat unfortunate. In the programming context, it is customary to think of :: as a list constructor. Hence one would like to think of any well-formed term in which :: appears as the outermost constructor as a list. This viewpoint is manifest in most recursive programs on lists that come to mind. However, the correctness of such programs depends on the purported 'list' object having the appropriate structure. Under this requirement, the program would function as expected when given the term (2:: "abc": nil) but not when provided the term (2:: "abc"). The ability to detect such an error prior to running the program is dependent on a typing system

that ensures that a 'list term' has the appropriate structure. In particular, the typing system should deem the expression (2 :: "abc") as ill-formed.

Similar arguments can be made in conjunction with predicate definitions. The standard example is that of append, the definition of which in $\lambda Prolog$ syntax is the following:

```
append nil L L. append (X :: L1) L2 (X :: L3) \leftarrow append L1 L2 L3.
```

Now the intended purpose of this definition is to describe a relation between lists. However, the typing of append that is currently under consideration permits a derivation for the query

```
append nil 3 3?
```

to be constructed successfully. The inappropriateness of posing this query can be detected and a better documentation can be provided for the program above if a distinguished sort for lists is used and the type of append is defined accordingly. Revising the typing of the constants in the following manner in fact 'solves' the various problems discussed:

```
2: i "abc": i nil: list
::: i \rightarrow list\rightarrow list append: list \rightarrow list\rightarrow list\rightarrow o
```

Notice in particular that the expression (2 :: "abc") is no longer a well-formed term. Although the ability to define new sorts appears useful, there is still a question concerning the purpose of type constructors. However, a justification for these is not hard to provide. Under the present typing, :: is a heterogeneous list constructor: the list (2 :: "abc" :: nil), for instance, has as its elements some objects that are best thought of integers and others as strings even though the present typing conflates this distinction. Now, there are contexts where it is useful to enforce homogeneity

on lists. Consider, for instance, the predicate sum_of_list that is defined as follows:

```
sum_of_list nil 0.
sum_of_list (X :: L) N \leftarrow sum_of_list L N1, add X N1 N.
```

We assume here that add is a predicate that 'performs' the addition of integers. We observe that add will perform as expected only when its arguments are integers. Thus, the query

```
sum\_of\_list (2 :: "abc" :: nil) N?
```

will eventually lead to an error. It may be observed, in general, that a query using sum_of_list is properly posed only if the first argument is a list of integers. The important point, of course, is to be able to detect this even as the query is written down.

It is useful to consider the typing changes that permit such errors to be detected at the time of compilation. To begin with, a distinction must be made between terms that denote integers and those that do not; the characterization of situations under which the predicate add is appropriately used depends on such a distinction. One solution is to dispense with the sort i and to introduce two new sorts int and string. An additional requirement is to provide a means for determining that a given list consists only of integers. Perhaps the simplest solution to this is to instead declare the type of the list constructors as follows:

```
nil : listint.
::: : (int \rightarrow listint \rightarrow listint).
```

However, this solution is not quite satisfactory. There may be a need to construct lists of other kinds of objects, e.g., strings. The modified typing of our list constructors precludes their use in constructing these lists. Although new constructors can be provided, this prevents a common definition of list processing predicates that do not depend on the type of the list elements. As a specific example, separate versions of append would have to be defined for lists of integers and strings.

A rather satisfactory solution to the problem of ensuring type correctness can be provided by using type constructors and type variables. Assuming that *list* is a unary type constructor, the following typing might be used for the constants in question:

```
2 : int "abc" : string nil : (list A) :: : A \rightarrow (list A) \rightarrow (list A) append : (list A) \rightarrow (list A) \rightarrow o
```

Under this typing, we observe that both (2 :: 2 :: nil) and ("abc" :: "abc" :: nil) are well-formed terms while (2 :: "abc" :: nil) is not. The types of the two well-formed terms are (list int) and (list string), respectively. Given the type of append, both terms could, under appropriate circumstances, be provided as arguments to it. Finally, the type of sum_of_list can be defined to be $((list\ int) \to int \to o)$. The query

```
sum\_of\_list (2 :: "abc" :: nil) N?
```

can be recognized to contain an ill-formed term at the time of compilation. The typing of sum_of_list will similarly permit the following query to be flagged as erroneous:

```
sum_of_list ("abc" :: "abc" :: nil) N?
```

We have thus far attempted to argue that the typing system of λ Prolog permits distinctions to be made between terms that are useful in developing correct and well-documented programs. Going in the other direction, we observe that the polymorphism present in the system permits even the differences based on the functional hierarchy to be collapsed up to a point. The essential idea is to define the type of every symbol as a variable. Thus, the typing

2:A "abc": A nil: A ::: A append: A

allows expressions such as $(2\ 2)$, ("abe" append), (append append), etc., to be considered well-formed. Looking at an expression such as $(2\ 2)$, one might be tempted to conclude that the distinction between an object and a function has vanished completely under this typing. However, this view is somewhat misleading. As explained in the previous subsection, the type declaration for 2 is in reality an abbreviation for a family of declarations. Two different constants from the collection so defined are intended to be used in the term $(2\ 2)$; the choice of these constants is further constrained by type considerations. The computational machinery underlying λ Prolog is cognizant of this requirement. It is often able to instantiate the types of polymorphic symbols to the extent required for computation, thereby maintaining the facade of typelessness. However, there are situations where outside help is required in determining instantiations for types. Such problems arise principally in the course unification, and will be discussed in detail towards the end of this section.

3.3 The Nature of the Polymorphism in λ Prolog

The notion of polymorphism, as opposed to *monomorphism*, corresponds to a situation in which some values and variables can have more than one type. In the programming context there is a particular interest in polymorphic functions, *i.e.*, functions whose arguments can have more than one type. Truly monomorphic languages turn out to be inconvenient to program in and, as pointed out in [1], even the most conventional programming languages manifest some form of polymorphism.

It is already clear that $\lambda Prolog$ is a polymorphic language. What is less clear is the nature of its polymorphism from a programming perspective. Strachey [28] distinguished between two major kinds of polymorphism: parametric polymorphism that is obtained when a function works uniformly on a range of types and ad hoc polymorphism that is obtained when a function works in somewhat unrelated ways on several different types. This classification is refined further in [1] to provide the dichotomy between universal polymorphism and ad hoc polymorphism. At an implementation level, the distinction amounts to the following: a universal polymorphic function executes the same code for arguments of any admissible type, whereas an ad hoc polymorphic function executes different code for each type of argument.

It is evident that $\lambda Prolog$ supports a form of universal polymorphism. The simplest manifestation of this is the *append* predicate encountered earlier in this section. This predicate can be used to append lists of objects of arbitrary type. Further, the code that is executed depends only on the structure of a list and remains the same regardless of the type of the elements in the list.

It turns out, however, that this is not the only form of polymorphism present in λ Prolog and that procedures can also manifest forms of ad hoc polymorphism. The main reason for this is that a procedure is typically defined by a collection of clauses and the clauses available at any particular invocation may depend on the specific types of the arguments. At the very extreme, the available clauses at one particular type may be disjoint from those at another type. An example of such a situation is provided by the following module that defines an 'interface' predicate.

```
module
               print.
kind
               string
                               type.
type
               print
                               A \rightarrow o.
               write_int
                              int \rightarrow o.
type
               write_list
                              (list A) \rightarrow o.
type
               write_string string \rightarrow o.
type
print (N : int) \leftarrow write\_int N.
print (L : (list A)) \leftarrow write\_list L.
print (S : string) \leftarrow write\_string S.
```

The code that gets executed when *print* is invoked depends on whether the argument is an integer, a list or a string, and entirely different code gets executed in each case.

It is argued in [1] that ad hoc polymorphism is only an apparent polymorphism, based on its categorization into overloading and coercion: overloading amounts to associating a common symbol with values that are monomorphic and in coercion the operation is ultimately monomorphic. Ad hoc polymorphism in λ Prolog has several features in common with overloading, and hence there is something to this argument even in this context. However, it is difficult to sustain the argument in its entirety. In particular, the manner in which procedures are defined permits the parametric and ad hoc polymorphism to be mixed in a way that makes a separation of the two difficult. Consider, for example, the following modification to the definition of append, assuming that a and b are of type i:

```
append (1 :: nil) (2 :: nil) (1 :: 2 :: nil).
append (a :: nil) (b :: nil) (a :: b :: nil).
append nil L L.
append (X :: L1) L2 (X :: L3) \leftarrow append L1 L2 L3.
```

Although some part of the code for append depends on the type of the elements of the lists being processed, there is also a part to this code that is independent of this. This aspect has ramifications from the perspective of implementations of the language. Consider, for instance, the invocation to append that appears in the body of the last clause. The code that is available for solving it evidently depends on the type of the list that is being processed and can only be determined

at run-time. Further, determining the available code requires type information to be transmitted across clause invocations. To appreciate this fact, one may consider solving the query

```
append (b :: L1) L2 (b :: L3)?
```

We see now that the query (append L1 L2 L3)? that arises in the process may erroneously be solved by the first clause if the type constraints on L1, L2 and L3 are not present at run-time. The question naturally arises as to whether there are situations when the need for such information can be eliminated. The notion of type-generality that is discussed in [5] for the case of first-order programs effectively amounts to excluding ad hoc polymorphism to achieve this end.

3.4 Types and Unification

The informal description of computation in the previous section alluded to differences between the operations of unification needed in Prolog and λ Prolog. One particular difference arises from the fact that terms in λ Prolog are typed. The consequences of typing are actually manifest even at the level of first-order terms. As an example, let a and b be constants of type i and let ft be a functor of type $A \to B \to i$ and consider unifying the two terms (ft X Y) and (ft a b). These terms have the same type so it makes sense to try to unify them even in the typed context. However, the existence of a unifier for these terms depends on the types of the variables X and Y. If these types are i, then the two terms have a unifier that is similar to the one obtained by ignoring the types. However, if the types are distinct from i then the terms are not unifiable. Looking at this issue from a different perspective, the conclusion is not unexpected. The constant ft that has type $(i \to i \to i)$ is really distinct from the one that has type $(int \to int \to i)$ and terms that have these different constants as their heads should not be unifiable. This observation can be given a more formal content: unification of typed first-order terms can be shown to be equivalent to unification for untyped terms obtained by adding the type of each functor as an additional argument of the function term. It follows from this that unification for typed first-order terms has certain properties, such as the existence of most general unifiers, that are similar to those for the corresponding untyped unification.

Although types might determine the existence of unifiers for first-order terms, the structure of these unifiers is in a certain sense independent of the types. This picture changes when one considers higher-order terms. In this context, the set of unifiers cannot be characterized by a most general unifier since a pair of terms may have unifiers that cannot be obtained via substitutions from one another. However, it is possible to consider a (perhaps infinite) set of unifiers that together subsume all other unifiers; such a set is called a complete set of unifier in [7]. The types that annotate the atoms in a term have a significant effect in determining the members

of such a set. For the purpose of illustration, let us consider the unifiers for the terms $(g\ a)$ and $(F\ X)$ assuming g has the type $i\to i$ and under different typings for F. To begin with let F have the type $int\to i$. There is evidently a most general unifier in this case, given by the substitution of the term $(Y\setminus (g\ a))$ for F. If the type of F is $i\to i$ on the other hand, there are three incomparable unifiers that subsume all others. These are given by the substitutions

$$\{\langle F, Y \setminus Y \rangle, \langle X, (g \ a) \rangle\}, \{\langle F, Y \setminus (g \ Y) \rangle, \langle X, a \rangle\}, \text{ and } \{\langle F, Y \setminus (g \ a) \rangle\}$$

The point to be observed is that changing a sort in the type of F, *i.e.*, replacing int by i, makes a significant difference with regard to the possible unifiers. Replacing the atomic type by a function type can similarly alter the set of unifiers. If F has the type $(int \rightarrow i) \rightarrow i$ then

$$\begin{array}{l} \{ \langle F, Y \setminus (g \ (Y \ Z)) \rangle, \langle X, (W \setminus a) \rangle \}, \\ \{ \langle F, Y \setminus (Y \ (Z \ Y)) \rangle, \langle X, W \setminus (g \ a) \rangle \}, \\ \{ \langle F, Y \setminus (g \ a) \rangle \} \}. \end{array}$$

is a complete set of unifiers and if the type of F is $(i \to i) \to i$ then such a set of unifiers actually becomes an infinite one.

The above discussion prompts several observations with regard to types in λ Prolog. Perhaps the first point to be noted is that types have a fundamental role to play within the underlying computational framework of the language, given the manner in which they affect the issue of unifiability and also the sets of unifiers. This is, of course, quite distinct from their role in detecting program errors and in a certain sense also from their function in disambiguating overloaded operations³. A particular consequence of this purpose of types is that they must adorn terms at run-time at least in principle; they are needed for determining the right computations. There is, nevertheless, the question of whether there are situations in which type information can be dispensed with at run-time even from the perspective of unifiability. The property of groundness of arguments described in [5] provides a partial answer for the case of first-order programs. The essential observation that is utilized in describing this property is that types affect the issue of unifiability of first-order terms without affecting the actual unifiers themselves. Thus, if the structures of the terms themselves contain enough information to answer the question of unifiability, types can be dispensed with. It appears that the results of [5] can be strengthened and that they might be practically useful for first-order programs. However, the distinct manner in which types affect higher-order unification requires the development of alternative techniques for this case.

A second observation is that types have a beneficial role to play even within the context of defining computations. A detailed understanding of this aspect requires

³Examples can be provided to distinguish the role of types in determining unifiability from their role vis-à-vis overloading. The assimilation of the latter within the former is arguable for first-order programs but is less tenable in the higher-order case.

a study of the higher-order unification procedure of [7], but the underlying intuition can already be grasped. Consider the problem of unifying the untyped terms $(F \ X)$ and $(g \ a)$, should this arise in some programming context. Clearly, these unifiers must include all those that can be obtained by adorning the two terms with types. The differences in the sets of unifiers that were obtained above by considering only a few different typings is convincing evidence of the impossibility of this task. This picture is considerably altered when types are present on these terms. In particular, the types structure the search for unifiers in a way that makes it a manageable process.

The final observation concerns the interaction between polymorphism and unification. As we noted in the last section, the type variables in a program clause must in theory be instantiated by closed types before the clause is used in solving a query. Unfortunately there may not be enough information for determining the desired type instance at the time the clause is invoked. A strict enforcement of this requirement can therefore be computationally costly. It turns out that the determination of type instance can in practice be delayed till enough information is available to identify it uniquely. The only place where type variables might need to be instantiated without delay is in fact within (higher-order) unification. To appreciate why this might be necessary, we might consider unifying the terms $((F : (A \rightarrow i)) X)$ and $((q : (i \rightarrow i)) a)$. We have observed that the unifiers of interest vary considerably depending on the choice of type for A. Some progress can be made even in this context without enforcing a type instantiation. For instance, we see that substituting $(Y \setminus (q \ a))$ for F unifies the terms under consideration regardless of how A is instantiated. The unification procedure can actually be structured so as to find such unifiers without requiring type instantiations. A detailed analysis of this issue is beyond the scope of this paper; the interested reader is referred to [19] for some discussion on it. However, we note that there is a limit to which the delaying process can be taken. In particular, the 'projection' substitution of [7] sometimes requires a decision of whether to instantiate a type variable with a function type or an atomic type. A procedure that is complete must try both possibilities. This is often impractical. The more viable options in such cases are to exercise the choice in favor of an atomic type or to indicate a run-time error.

4 Approaches to Type Reconstruction

In a statically typed language, one must face a succession of fundamental questions regarding the nature of the type system. Perhaps the most basic is the decidability of type-checking. We have to devise an algorithm that, given a program, decides if it is well-typed, as defined by the typing rules. Obviously, languages with an undecidable type-checking problem are suspect, since one cannot build a compiler that accepts exactly the well-formed programs.

Of great practical significance is the usually more difficult problem of type reconstruction: given a program with only partial type information, fill in the missing types in such a way that the resulting program is well-typed, or issue an error message if this is impossible. Whether type reconstruction is possible and feasible depends crucially on the precise definition of partially typed expression.

Now in general there may be more than one possible way of reconstituting omitted types, and the question then arises as to what the 'right' choice of type might be in such a situation. A matter of some interest is that of whether a finite representation can be found for all the possible types that can be assigned to a given expression. If such a representation exists, it is referred to as a principal type of the expression. A type system is said to have the principal type property if every partially typed expression has a principal type. The existence of principal types has important practical consequences. Firstly, the user can be informed about the type of an expression, as it was inferred during type reconstruction. Secondly, a type reconstruction system can use these principal types during its execution, thereby eliminating the need to 'guess' types and consequently also the need to backtrack at any stage.

4.1 Polymorphism for Constants and Variables

Within the functional language ML, there is a need to carefully distinguish identifiers that may be genuinely polymorphic from those that may not. As a simple example, consider

let
$$id = \lambda x.x$$
 in $(id 1, id \text{ true})$

In this example, the identifier id may be assigned the type schema $\forall A. A \rightarrow A$, with A being instantiated to int and bool at the two occurrences of id in the body of the let-expression, respectively. Such an assignment is justifiable in the sense that the well-typedness of the the expression under it guarantees the absence of type errors at run-time. On the other hand, no assignment of a type acceptable within ML would permit the following expression to be considered well-typed:

val
$$g = \lambda f$$
. $(f 1, f \text{ true})$

It may be suggested that a possible typing for g is $\forall A. (A \to A) \to int \times bool$, but this has problems. The expression $g(\lambda x. x+1)$ would be be deemed to be well-typed under this assignment, but evaluating it would require computing true +1, thereby leading to a run-time 'type' error. A better type for g would be $g: (\forall A. A \to A) \to int \times bool$, in effect forcing the argument to be a polymorphic function, but this would be outside the type system for ML, since it contains an embedded universal quantifier.

To summarize the above discussion, λ -abstracted variables may not be used polymorphically in ML whereas let-abstracted variables may. Similar reasons may

be advanced for not allowing variables in λ Prolog to be used polymorphically. For example, consider the program

```
\begin{array}{lll} type & even & int \rightarrow o. \\ type & empty & list \ B \rightarrow o. \\ type & q & A \rightarrow o. \\ q \ X \leftarrow even \ X, \ empty \ X. \end{array}
```

If we were allowed to assign, say, the type schema $\forall A. A$ to X and to instantiate this to *int* and *list* B at the two occurrences of X in the body of the clause, the program would be considered well-typed. Further, a query such as

```
q 0 ?
```

would be well-typed, but would lead in the course of evaluation to the ill-typed subgoal empty 0.

The problem with permitting variables to be polymorphic may, upon reflection, be seen to arise from a dual set of facts: variables may be instantiated in the course of evaluation and the type system is not strong enough to guarantee that the instantiating expression has itself the desired polymorphic type. This problem cannot arise with constants since these are never instantiated. Constants may therefore be permitted to be used polymorphically.

4.2 Type Reconstruction for Variables

The types of variables in a $\lambda Prolog$ program may be omitted without creating undesirable ambiguity in the definition of the clauses. The norm in determining the omitted types may be taken to be that of ensuring maximal applicability for the clauses in which the variables occur. Assuming that the types of constants are all explicitly declared, the fact that variables cannot be used polymorphically guarantees that this norm can be met: there is an assignment of types all instances of which yield a well-typing and that subsumes all other assignments that yield a well-typing. The problem of finding such types for variables can be solved by using first-order unification in the same way as is done in the context of ML [14, 3]. This can easily be checked through the inference rules in Figure 1.

4.3 Type Reconstruction for Constants

Recalling the discussions in Subsection 3.1, there appear to be two different situations in which the reconstruction of types for constants is required. In the first case, the type schema to be associated with a constant may not be known from the presentation of the program. In the second case, the type schema may be known but the intended type instance at a particular occurrence of the symbol may not

be provided. As in the case with variables, the latter omission creates an ambiguity that can be resolved easily. The principle of ensuring maximal applicability may once again be taken to be the norm in determining the omitted instantiation. Further the problem of finding the appropriate instantiation under this assumption can again be solved by using first-order unification.

The issue of type reconstruction becomes more problematic if type declarations for constants are also omitted. This may appear surprising, given that the general type reconstruction problem is well understood in the context of functional languages. The main source of the complication is that constants may be polymorphically typed. Within a language like ML, there are two categories of objects that can be so typed: the *let* bound identifiers and the data constructors. The types for data constructors in such a language are always declared explicitly. As for the types of *let* bound identifiers, these are determined by the most general type assignable to the expression they are bound to, and calculating these types is a relatively simple task. In λProlog , in contrast, there are polymorphically typed constants whose types may have been omitted and that do not have an associated 'definition.' The types of these symbols must be reconstructed from their uses instead.

At the very outset, we observe that the type system under consideration does not have the principal type property when types of polymorphically typeable objects may remain undeclared. To take the particular case of λProlog , consider the following program clause in which the type of q is unspecified:

q 1.

This program is well-typed under the assignment of the type $int \to o$ or $\forall A. A$ to q. However it is not well-typed under assignments such as $q: i \to o$ and q: o. Now there is no expression in our type language that subsumes the acceptable assignments for q but not the unacceptable ones. It follows, thus, that a principal type assignment does not always exists when we restrict ourselves to looking for expressions from our type language.

The above discussion actually raises a question about the very utility of determining a principal type assignment — possibly in an auxiliary type language — even if such a notion were to be sensible in the context of interest. If a program is typeable at all, then it must be so under the assignment of the type $\forall A.$ A for each of the undeclared constants. This is so because any type that might be required for a constant c is an instance of $\forall A.$ A. However, such an assignment gives virtually no information with regard to what are the intended types for the constants and serves mainly to determine the consistency of the types explicitly provided. Since a principal type assignment must subsume the kind of typing just discussed, it would in turn contain little useful information about a given program. Undisputed, however, is the import of the notion of typeability: can we restore types to a partially typed program so that the result is well-typed. Presumably a compiler would have to accept partially typed programs that are typeable, and reject programs that cannot

be well-typed.

There is in fact a reason for even suspecting the idea of a principal type within the $\lambda Prolog$ context, at least if interpreted naively. In a language like ML, the assignment of a type to an expression may be interpreted as stating a property of that expression. The principal type assignment under this view expresses the strongest property of the expression that can be stated within the type system. It is important to note that, in ML, the meaning of a program is independent of the particular assignment. However, this is not the case in $\lambda Prolog$ if one construes meaning as the execution behavior of programs. As an example, consider the program

```
type nil list A.

type append (list A) \rightarrow (list B) \rightarrow (list C) \rightarrow o.

append nil L L.

append (X :: L1) L2 (X :: L3) \leftarrow append L1 L2 L3.
```

This program is well-typed under the assignment of either $\forall A. A \rightarrow (list \ A) \rightarrow (list \ A)$ or $\forall A. \forall B. \ B \rightarrow (list \ B) \rightarrow (list \ A)$ to the list constructor ::. Now, the behavior of the program depends on the particular choice made. Thus, consider the query

```
append (1 :: nil) (a :: nil) L?
```

assuming that the type of a is i. This query is well-typed under either assignment but it results in a failure under the first typing and succeeds under the second. Notice further that the first type is an instance of the other. Thus, the choice of a type affects the meaning of the program, even if one is only choosing a more general type than a given one.

The problem of type reconstruction in $\lambda Prolog$ thus seems to involve picking one of several competing type assignments that is useful in determining the 'correctness' of the program and that simultaneously guesses at the intended meaning of the program. The criterion for making of such a choice must clearly be dictated by pragmatic considerations and not logical ones. As a first pass at such a criterion, we might offer the following: among all possible typings for a constant, we pick the one that is most general and has the characteristic that all instances of it are also acceptable typings. As an example, let us consider once again the program clause

q 1.

Using the criterion just enunciated, the type that would be picked for q would be $int \to o$; the type $\forall A. A \to o$ would, for instance, not be chosen because it has as an instance the unacceptable type $i \to o$. As another example, consider the clause

qX.

Here one would choose the type $\forall A. A \rightarrow o$ for q.

Unfortunately the criterion suggested above is too simplistic and does not work in the following sense: there are several situations in which a program is well-typed under type assignments none of which satisfy the criterion. The typical scenario is one where multiple occurrences of a constant dictate choices that are not simultaneously reconcilable. An example of this sort is provided by the program

q 1. q nil.

Here, the first occurrence of q suggests the type $int \to o$ whereas the second suggests the type $\forall A.\ list\ A \to o$ for q. These are not unifiable and, hence, there is no expression within our type language that satisfies both requirements. However, there exist a number of assignments under which the program is well-typed, such as $q: \forall A.\ A \to o$ or $q: \forall A.\ \forall B.\ A \to B$.

Before proceeding further in this discussion, it would be useful to spell out some principles that we would like a type reconstruction algorithm for λP rolog to follow. The above example suggests one: a type assignment should be suggested for every program that can be well-typed. There is actually another principle that can be distilled from this example. From examining it carefully, we see that type reconstruction for constants cannot be fully incremental: we need to see all occurrences of q before we can decide on a satisfactory type. It is, of course, not possible to postpone a decision on the type of a constant forever, and some predetermined program boundaries must be adhered to in the type reconstruction process. In summary, the type reconstruction algorithm may be required to satisfy the following conditions:

- 1. Analysis of occurrences of constants for the purpose of type reconstruction should be confined to a single program module or single query.
- 2. A type assignment should be suggested for a program that can be well-typed.
- 3. No type assignment should be suggested for a program that cannot be well-typed.

It is important to note that these requirements do not by themselves completely determine the outcome of type reconstruction. We have already observed that a trivial and correct solution would be to assign the type $\forall A.$ A to every undeclared constant in a program that can be well-typed. The problem with this solution is that the resulting types of constants will almost always be too general (from the intuitive point of view) and thus not very useful in detecting incorrect uses of a constant outside of the defining module. If type-checking is to satisfy the purpose of discovering (many) incorrect programs prior to execution, some intuitively acceptable way of further restricting the assigned types has to be found.

Returning now to the earlier criterion, we would like to refine it so that it satisfies the principles discussed above. Let us call a type schema σ minimal for a constant

c if it is a legal assignment in the sense that the program can be well-typed under it and if, in addition, no instance of σ is also a legal assignment for c. The canonical type of a constant c is then the least upper bound of all minimal types of c. In the examples above, the canonical type of an expression appears to accord well with what we might think of as its correct type. Thus, consider the program clause

q 1.

Here, any type that can be instantiated to $int \to o$ is a legal type for q. There is only one minimal type, namely $int \to o$, the desired type for q. Alternatively consider

qX.

Here, $int \to o$, $list\ int \to o$, ... are all minimal types. The least upper bound of all of them is $\forall A.\ A \to o$, again the desired result. Finally, consider

q 1. q nil.

Here, neither $int \to o$ nor $list int \to o$ are legal types for q. Only types more general than $\forall A.\ A \to o$ are legal. Thus $\forall A.\ A \to o$ is the only minimal type, and thus the least upper bound of all minimal types.

Thus, it may appear that canonical types are a reasonable choice for what a type reconstruction algorithm should produce. Unfortunately there are problems with this notion. One problem is that the least upper bounds calculated separately for each constant may not be compatible. Consider

 $\begin{array}{l} \mathbf{q} \leftarrow \mathbf{c} \ \mathbf{X}, \, \mathbf{d} \ \mathbf{X}. \\ \mathbf{c} \ 1. \\ \mathbf{d} \ \mathbf{nil}. \end{array}$

In this case, the least upper bound for the type of c would be $int \to o$ (since we can pick $d: \forall A. \ A \to o$ and X:int), while the least upper bound for the type of d would be $\forall A. \ list \ A \to o$ (since we can pick $c: \forall A. \ A \to o$ and $X:list \ A$), but we cannot have $c:int \to o$ and $d:list \ A \to o$ simultaneously, since there would be no legal type for X. (Recall that variables can only be assigned simple types, not type schemas).

One way of resolving this problem is to pick the least upper bound that is compatible with all other choices one might make. Thus, in the case above, we may pick the assignment $c: \forall A.\ A \to o$ and $d: \forall A.\ A \to o$. However, the making of such a choice is *not* stable under extensions to the program in a rather subtle way. For example, if we add one clause to the earlier program to obtain

```
q \leftarrow c X, d X.
c 1.
d nil.
d 2.
```

then d is already forced to be polymorphic $(d : \forall A. A \rightarrow o)$, thus 'releasing' c, that can now be typed with $int \rightarrow o$.

Perhaps worse than the previous problem is the fact that a canonical type may not exist even when the program itself can be well-typed. Consider, for instance, the program clause

qq.

This is well-typed under the assignment $q: \forall A. A \rightarrow o$. However this type for q is not minimal. Consider $q: \forall A. (A \rightarrow o) \rightarrow o$. This is also a solution, and an instance of the first solution. We can repeat this argument, yielding $q: \forall A. ((A \rightarrow o) \rightarrow o) \rightarrow o$, etc. Thus, in such a case, there are no minimal types, since below every solution there lies another, less general solution.

This calls into question the whole enterprise of making a canonical choice of some sort between the possibilities. Instead, we will try to develop a practical algorithm that determines whether a given program is typeable, that suggests an acceptable type assignment that is acceptable in most situations in which this is the case and that is computationally tractable.

4.4 An Algorithm for Type Reconstruction

The algorithm is based on the observation that, unless there is a non-unifiability of some sort (cycle or constant clash), unifying the type constraints imposed on the constant at their various occurrences will result in a canonical type. If there is a clash, as in the types for q when analyzing q 1 and q nil, we will need to find the least general type that can be instantiated to the candidate types, in this case to $int \to o$ and $list B \to o$. The problem of finding such a generalization is known as anti-unification in the literature [23, 26]. In the example of interest, anti-unification would result in the type $A \to o$.

Anti-unification, while simpler than unification, is not entirely trivial. Consider, for example, the case when we are given q 12 and q (2 :: nil) (1 :: nil). Analyzing these expressions, we find that that the type of q must be instantiable to both $int \to int \to o$ and list $int \to list$ $int \to o$. The anti-unifier for these two expressions is $A \to A \to o$. Note that we reuse the same type variable A. If we had been given the expressions q 1 nil and q nil 2 instead, the least upper bound for the types of q would have been $A \to B \to o$.

The main steps in the algorithm are the following:

- Type reconstruction for variables is performed through unification, as in ML.
 A fresh type variable is generated each time the type of an undeclared constant c is required. All the type variables that have been used for the type of c are kept track of as constraints on the type schema for c.
 - If this phase fails, we know that the program is *not* typeable. The algorithm consequently issues an appropriate error message.
 - If this phase succeeds, we know that the program is typeable. It remains to suggest types for the undeclared constants.
- 2. For each constant c, an attempt is made to unify all the constraints on c that were accumulated during phase 1.
- 3. If this succeeds, all free variables in each type declaration are abstracted over to obtain a type schema for each constant. In this case, each type generated is guaranteed to be canonical, *i.e.*, they are the least upper bounds of all minimal types that lead to a well-typed program.
- 4. If step 2 fails, there must have been a disagreement. At the place where the type constraints for a constant disagree (because of a constant clash or cycle), the disagreement is replaced with a type parameter a that is not subject to instantiation and the disagreement is recorded. If the same disagreement is encountered again later, the same parameter a is substituted for it. This idea is the essence of algorithms for anti-unification. Finally all the free variables and parameters are abstracted over to obtain the type schemas for the constants.

In the case that disagreements are encountered, the algorithm just described is not guaranteed to obtain a canonical type for each constant. As we have demonstrated earlier in this section, canonical types may not exists or may not be unique, so this should come as no surprise.

We illustrate the algorithm through a few simple examples. Consider first the program

```
type nil list A.

type :: A \rightarrow list A \rightarrow list A.

append nil L L.

append (X :: L1) L2 (X :: L3) \leftarrow append L1 L2 L3.
```

The first step, when applied to this program, yields the constraints

```
append \sim list A1 \rightarrow B1 \rightarrow B1 \rightarrow o.
append \sim list A2 \rightarrow B2 \rightarrow list A2 \rightarrow o.
append \sim list A2 \rightarrow B2 \rightarrow list A2 \rightarrow o.
```

from the three occurrences of append, respectively. Unifying these constraints succeeds and we infer the declaration as

type append list
$$A \rightarrow list A \rightarrow list A \rightarrow o$$
.

If the types of *nil* and :: were also unknown, the behavior would be different. We would then obtain the constraints

```
\begin{array}{l} \mathrm{append} \sim \mathrm{A1} \rightarrow \mathrm{B1} \rightarrow \mathrm{B1} \rightarrow \mathrm{o.} \\ \mathrm{append} \sim \mathrm{A2} \rightarrow \mathrm{B2} \rightarrow \mathrm{A3} \rightarrow \mathrm{o.} \\ \mathrm{append} \sim \mathrm{A4} \rightarrow \mathrm{B2} \rightarrow \mathrm{A5} \rightarrow \mathrm{o.} \\ \mathrm{nil} \sim \mathrm{A1.} \\ :: \sim \mathrm{B3} \rightarrow \mathrm{A4} \rightarrow \mathrm{A2.} \\ :: \sim \mathrm{B3} \rightarrow \mathrm{A5} \rightarrow \mathrm{A3.} \end{array}
```

and, again by unification, we infer

$$\begin{array}{lll} \text{type} & \text{nil} & B. \\ \text{type} & & \vdots & A \rightarrow B \rightarrow B \rightarrow o. \\ \text{type} & \text{append} & B \rightarrow B \rightarrow o. \\ \end{array}$$

Note that, because each type is implicitly quantified in each declaration, the original relationship between B in the three types is lost. However, if the programmer would like to extract declarations from this output, the relationship might still be useful. One might imagine a different form of abstraction, where free type variables are abstracted over the whole module, leading instead to the inferred declarations

```
\begin{array}{ccccc} kind & a & type. \\ kind & b & type. \\ type & nil & b. \\ type & :: & a \rightarrow b \rightarrow b \rightarrow o. \\ type & append & b \rightarrow b \rightarrow b \rightarrow o. \end{array}
```

Now we consider an example where unification does not succeed.

```
doc append "Appends lists".
doc doc "An overloaded documentation predicate".
```

Given the type of append, the constraints generated take the form

$$\begin{array}{l} \operatorname{doc} \sim (\operatorname{list} \, A \to \operatorname{list} \, A \to \operatorname{o}) \to \operatorname{string} \to \operatorname{o}. \\ \operatorname{doc} \sim \operatorname{B} \to \operatorname{string} \to \operatorname{o}. \\ \operatorname{doc} \sim \operatorname{B}. \end{array}$$

Unification fails because of a cyclic dependency. B is replaced by a parameter b (this argument must be polymorphic). This is then abstracted again in step 3 (there are no further constraints) and we obtain

$$type \qquad doc \qquad B \to string \to o.$$

Ad hoc polymorphism that is used to model overloading is the most frequent source of type-clashes that need to be generalized. Consider the simple overloading example from Section 3.3.

```
type write_int (int \rightarrow o).

type write_list ((list A) \rightarrow o).

type write_string (string \rightarrow o).

print N \leftarrow write_int N.

print L \leftarrow write_list L.

print S \leftarrow write_string S.
```

The constraints on *print* clash, leading to the desired type schema $\forall A. A \rightarrow o$ for *print*. Note that the first clause will only be selected when the argument to *print* is an integer, since unification respects types.

Finally, let us consider an example that shows why we perform anti-unification rather than always replacing disagreements with new parameters. Assuming a and b are of type i, consider

```
append (1::nil) (2::nil) (1::2::nil). append (a::nil) (b::nil) (a::b::nil). In this case, the constraints  \begin{aligned} \text{append} &\sim \text{list int} \to \text{list int} \to \text{list int} \to \text{o.} \\ \text{append} &\sim \text{list } i \to \text{list } i \to \text{o.} \end{aligned}
```

do not unify. The disagreement we obtain is between int and i at all three places where the types disagree, and we replace them with the same parameter a. This parameter is later abstracted and we obtain the expected type for append.

5 Conclusion

Types, when used in a prescriptive sense, are widely recognized to be a valuable aid in the development of correct programs. They function by enforcing minimal consistency requirements on acceptable programs and thereby precluding the writing of programs that will manifest certain kinds of errors during execution. However, the device of restricting the set of legal programs is a double-edged sword: an overzealous criterion may make it difficult to write programs for performing certain tasks and may, in some cases, even make this impossible. There is, thus, a delicate balance to be maintained in developing a type system that is to be used in a practical programming language. The eventual typing scheme should manifest a sufficient

degree of generality but must also have the capability of alerting the programmer to problems that would develop if the program were to be executed.

We have presented the type system of the language $\lambda Prolog$ in this paper and have used this context to discuss several issues pertinent to typing in logic programming in general. The particular type system used in $\lambda Prolog$ is based on the discipline of simple types extended to incorporate a form of polymorphism. The typing thus provided is interesting for several different reasons. It offers a useful means for detecting errors in the program during compilation, a fact that we have demonstrated through several examples. The presence of polymorphism allows for code reuse and overloading, thereby making the overall language a fairly flexible one for programming. Finally, it is only through the use of types that certain higher-order aspects of the language become possible: types are necessary if the language is to support higher-order data types and they also provide a sound logical basis for higher-order programming features such as the metacall construct.

While the type system of $\lambda Prolog$ has several endearing features, there are certain respects in which the descriptive power of the underlying type language can be strengthened. One particular possibility is the inclusion in the language of some device for forming unions of types. Assume, for instance, that we are to define a procedure p that is to operate on integers and lists of integers. The only possibility that exists using the current type language is to identify the type of p as $A \to o$, thus making it completely polymorphic. From the perspective of detecting errors, this is somewhat unfortunate. Under this typing it would be impossible to detect an erroneous attempt to invoke p on, say, a list of lists of integers.

Much work has been done within logic programming towards developing fairly expressive type languages in the first-order setting. These languages have variously included aspects of subtypes, union types, dependent types and recursively defined types. One particular proposal in this context is the language of regular trees [15, 16, 29] that, intuitively, permits descriptions of collections of terms to be formed by using operations such as union and recursion over terms. The main use of this kind of a language has been in conjunction with the idea of descriptive typing, the focus being on methods for inferring the success sets of predicates in a program⁴. There is, nevertheless, reason to believe that a language such as this one might also be useful in a prescriptively typed, higher-order context. Studying the feasibility and the practical benefits of this possibility are aspects we believe to be worth pursuing.

The discussions in this paper have exposed a curious characteristic of prescriptively typed logic programming languages, namely the need for types to be present at run-time. A requirement of this sort is worrisome from the perspective of practicality and it therefore seems necessary to comment on the overheads to be incurred

⁴However, see [29] for a curious exception. The latter part of the mentioned paper uses type declarations in the form of regular types to 'type-check' programs. The notion of type-checking used appears to include determining that types are not needed at run-time, and is thus distinct from the one discussed here.

due to typing. One aspect of the overhead involves the use of space: term representations must now include an extra component, namely the type annotations. Another aspect of the overhead concerns the time needed to perform type analysis. With regard to first-order programs, it has been observed that a run-time processing of types becomes unnecessary when these satisfy certain typing constraints [18, 5]. However, typing annotations must be maintained even in these cases since, in order to be technically correct, type information must accompany any answer given. In the general case, a close look at the typing regimen shows that a clever representation of types and a careful use of information present during compilation can considerably reduce the time and space overhead. Consider, for instance, the constructor: whose type is declared to be $A \to (list \ A) \to (list \ A)$. The type of any instance of this constant would differ from this 'skeleton' only in the instantiation for A. Further, it is only this instantiation that needs to be 'checked' and the necessary checking can often be compiled. These observations have been used in [8] to demonstrate that the first-order version of $\lambda Prolog$ can be implemented with very little actual overhead over the untyped language. Based on this experience, we believe it is possible to obtain the benefits of using a typed language without paying an exorbitant price in terms of performance.

6 Acknowledgements

The work of Nadathur was partially supported by NSF Grant CCR-89-05825 and Army Research Office Grant DAAL03-88-K-0082. The work of Pfenning was supported in part by the Office of Naval Research under contract N00014-84-K-0415 and in part by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 5404, monitored by the Office of Naval Research under the same contract. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of DARPA or the U.S. Government.

References

- [1] Cardelli, L. and Wegner, P. On understanding types, data abstraction, and polymorphism. ACM Computing Surveys 17 (4), 1985, 471 522.
- [2] Church, A. A formulation of the simple theory of types. *Journal of Symbolic Logic* 5 (1940), 56 68.
- [3] Damas, L. and Milner, R. Principal type schemes for functional programs. Proceedings of the Ninth ACM Symposium on Principles of Programming Languages, Albuquerque, 1982, 207 212.

- [4] Gordon, M.J., Milner, A.J. and Wadsworth, C.P. Edinburgh LCF, Springer Verlag, 1979.
- [5] Hanus, M. Polymorphic higher-order programming in Prolog. Proceedings of the Sixth International Logic Programming Conference, Lisbon, June 1989, MIT Press, 382 – 398.
- [6] Hindley, J. R., and Seldin, J. P. Introduction to combinators and λ-calculus, Cambridge University Press, Cambridge, 1986.
- [7] Huet, G. P. A unification algorithm for typed λ -calculus. Theoretical Computer Science 1 (1975) 27 57.
- [8] Kwon, K., Nadathur, G. and Wilson, D.S. Implementing logic programming languages with polymorphic typing. Duke Technical Report CS-1991-39, October 1991.
- [9] Miller, D. Lexical scoping as universal quantification. Proceedings of the Sixth International Logic programming Conference, Lisbon, June 1989, MIT Press, 268 – 284.
- [10] Miller, D. A logical analysis of modules in logic programming. The Journal of Logic Programming 6, 1 & 2 (1989), 79 109.
- [11] Miller, D. Unification under a mixed prefix. Journal of Symbolic Computation (to appear).
- [12] Miller, D. and Nadathur, G. Higher-order logic programming. Proceedings of the Third International Logic Programming Conference, London, June 1986, Springer Verlag LNCS 225, 448 - 462.
- [13] Miller, D., Nadathur, G., Pfenning, F. and Scedrov, A. Uniform proofs as a foundation for logic programming. *Annals of Pure and Applied Logic* 51 (1991) 125 157.
- [14] Milner, R. A theory of type polymorphism in programming. *Journal of Computer and System Sciences* 17 (1978), 348 375.
- [15] Mishra, P. Towards a theory of types in Prolog. Proceedings of the 1984 International Symposium on Logic Programming, Atlantic City, February 1984, 289 299.
- [16] Mishra, P. and Reddy, U.S. Declaration-free type checking. ACM Symposium on Principles of Programming Languages, 1985, 7 21.

- [17] Mitchell, J.C. and Harper, R. The essence of ML. Proceedings of the Fifteenth Annual ACM Symposium on Principles of Programming Languages, San Diego, January 1988, 28 – 46.
- [18] Mycroft, A. and O'Keefe, R.A. A polymorphic type system for Prolog. Artificial Intelligence 23, 1984, 295 307.
- [19] Nadathur, G. A higher-order logic as the basis for logic programming. Ph.D. Dissertation, University of Pennsylvania, May 1987.
- [20] Nadathur, G. and Miller, D. An overview of λProlog. Proceedings of the Fifth International Conference and Symposium on Logic Programming, Bowen, K. and Kowalski, R., eds., MIT Press, 1988, 810 – 827.
- [21] Nadathur, G. and Miller, D. Higher-order Horn clauses. J. ACM, Vol. 37, No. 4, October 1990, 777 – 814.
- [22] L. Paulson, The foundations of a generic theorem prover. University of Cambridge Technical Report 130, March 1988.
- [23] Plotkin, G. D. A note on inductive generalization. *Machine Intelligence*, 5: 153 163, 1970.
- [24] Pyo, C. and Reddy, U.S. Inference of polymorphic types for logic programs. Proceedings of the North American Conference on Logic Programming, Lusk, E. L. and Overbeek, R. A., eds., MIT Press, 1989, 1115 1132.
- [25] Reddy, U.S. Notions of polymorphism for predicate logic programs. Proceedings of the Fifth International Conference and Symposium on Logic Programming, Bowen, K. and Kowalski, R., eds., MIT Press, 1988.
- [26] Reynolds, J. C. Transformational systems and the algebraic structure of atomic formulas. *Machine Intelligence*, 5: 135 151.
- [27] Robinson, J.A. A machine-oriented logic based on the resolution principle. J. ACM 12 (1965), 23 - 41.
- [28] Strachey, C. Fundamental concepts in programming languages. Lecture Notes for International Summer School in Computer Programming, Copenhagen, August 1967.
- [29] Yardeni, E. and Shapiro, E. A type system for logic programs. In *Concurrent Prolog: Collected Papers*, Volume 2, Shapiro, E. (ed), MIT Press, 1987, 211 244.
- [30] Zobel, J. Derivation of polymorphic types for Prolog programs. Proceedings of the Fourth International Conference on Logic Programming, Melbourne, 1987, MIT Press, 817 – 838.